Activity 1.1 : Neural Networks

Name: Santos, Tristan Neal and Espiritu, Maj
Section:CPE32S9

Objective(s):

This activity aims to demonstrate the concepts of neural networks

Intended Learning Outcomes (ILOs):

- Demonstrate how to use activation function in neural networks
- Demonstrate how to apply feedforward and backpropagation in neural networks

Resources:

- Jupyter Notebook

Procedure:

Import the libraries

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Define and plot an activation function

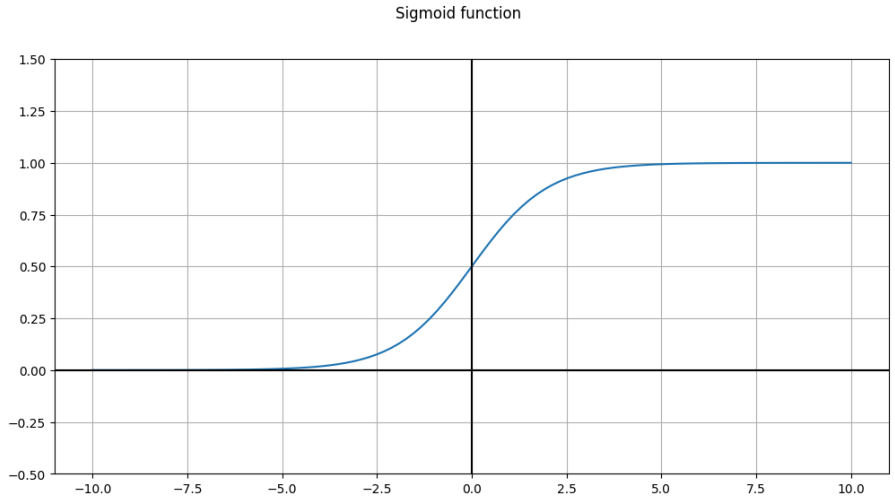
Sigmoid function:

$$\sigma = \frac{1}{1+e^{-x}}$$

σ ranges from (0, 1). When the input x is negative, σ is close to 0. When x is positive, σ is close to 1. At $x = 0$, $\sigma = 0.5$

```
## create a sigmoid function
def sigmoid(x):
    """Sigmoid function"""
    return 1.0 / (1.0 + np.exp(-x))
```

```
# Plot the sigmoid function
vals = np.linspace(-10, 10, num=100, dtype=np.float32)
activation = sigmoid(vals)
fig = plt.figure(figsize=(12,6))
fig.suptitle('Sigmoid function')
plt.plot(vals, activation)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.yticks()
plt.ylim([-0.5, 1.5]);
```

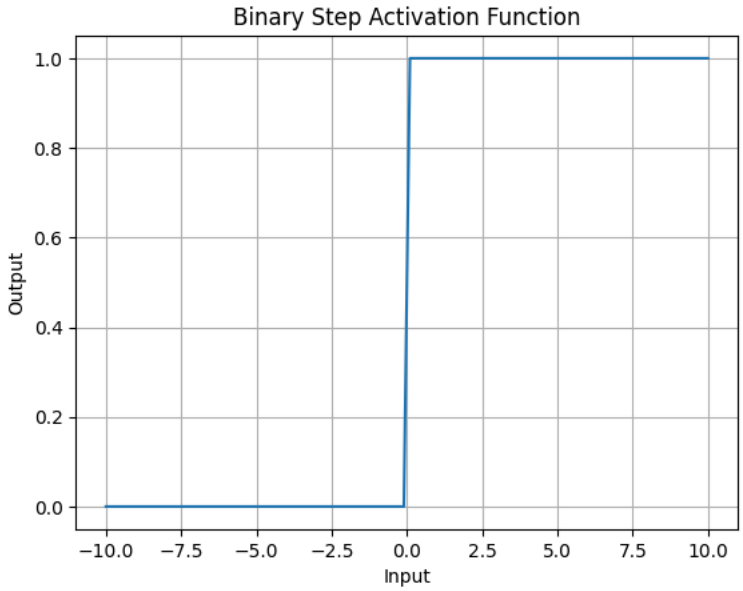


Choose any activation function and create a method to define that function.

```
class activation_function:

  def binary_step_function(self, x):

    return np.heaviside(x,1)
```

Plot the activation function

```
# Generate input values
x = np.linspace(-10, 10, num=100, dtype=np.float32)

# Apply the binary step activation function
y = binary_step(x)

# Plot the activation function
plt.plot(x, y)
plt.title('Binary Step Activation Function')
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid(True)
plt.show()
```



Neurons as boolean logic gates

OR Gate

OR gate truth table

Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

A neuron that uses the sigmoid activation function outputs a value between (0, 1). This naturally leads us to think about boolean values.

By limiting the inputs of x_1 and x_2 to be in $\{0, 1\}$, we can simulate the effect of logic gates with our neuron. The goal is to find the weights , such that it returns an output close to 0 or 1 depending on the inputs.

What numbers for the weights would we need to fill in for this gate to output OR logic? Observe from the plot above that $\sigma(z)$ is close to 0 when z is largely negative (around -10 or less), and is close to 1 when z is largely positive (around +10 or greater).

$$z = w_1 x_1 + w_2 x_2 + b$$

Let's think this through:

- When x_1 and x_2 are both 0, the only value affecting z is b. Because we want the result for (0, 0) to be close to zero, b should be negative (at least -10)
- If either x_1 or x_2 is 1, we want the output to be close to 1. That means the weights associated with x_1 and x_2 should be enough to offset b to the point of causing z to be at least 10.
- Let's give b a value of -10. How big do we need w_1 and w_2 to be?
  - At least +20
- So let's try out $w_1 = 20$, $w_2 = 20$, and $b = -10$!

```
def logic_gate(w1, w2, b):
    # Helper to create logic gate functions
    # Plug in values for weight_a, weight_b, and bias
    return lambda x1, x2: sigmoid(w1 * x1 + w2 * x2 + b)

def test(gate):
    # Helper function to test out our weight functions.
    for a, b in (0, 0), (0, 1), (1, 0), (1, 1):
        print("{}, {}: {}".format(a, b, np.round(gate(a, b))))


or_gate = logic_gate(20, 20, -10)
test(or_gate)
```

```
    0, 0: 0.0
    0, 1: 1.0
    1, 0: 1.0
    1, 1: 1.0
```

OR gate truth table

Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

Try finding the appropriate weight values for each truth table.

AND Gate

AND gate truth table

Input		Output
0	0	0
0	1	0
1	0	0
1	1	1

Try to figure out what values for the neurons would make this function as an AND gate.

```
# Fill in the w1, w2, and b parameters such that the truth table matches
w1 = 2
w2 = 2
b = -2.5
and_gate = logic_gate(w1, w2, b)

test(and_gate)
```

```
    0, 0: 0.0
    0, 1: 0.0
    1, 0: 0.0
    1, 1: 1.0
```

Do the same for the NOR gate and the NAND gate.

```
w1 = -1
w2 = -1
b = 0.5
nor_gate = logic_gate(w1, w2, b)

test(nor_gate)
```

```
    0, 0: 1.0
    0, 1: 0.0
    1, 0: 0.0
    1, 1: 0.0
```

```
w1 = -2
w2 = -2
b = 3
nand_gate = logic_gate(w1, w2, b)

test(nand_gate)
```

```
    0, 0: 1.0
    0, 1: 1.0
    1, 0: 1.0
    1, 1: 0.0
```

Limitation of single neuron

Here's the truth table for XOR:

XOR (Exclusive Or) Gate

XOR gate truth table

	Input		Output
0	0	0	
0	1	1	
1	0	1	
1	1	0	

Now the question is, can you create a set of weights such that a single neuron can output this property?

It turns out that you cannot. Single neurons can't correlate inputs, so it's just confused. So individual neurons are out. Can we still use neurons to somehow form an XOR gate?

```
# Make sure you have or_gate, nand_gate, and and_gate working from above!
def xor_gate(a, b):
    c = or_gate(a, b)
    d = nand_gate(a, b)
    return and_gate(c, d)
test(xor_gate)
```

```
 0, 0: 0.0
 0, 1: 1.0
 1, 0: 1.0
 1, 1: 1.0
```

Feedforward Networks

The feed-forward computation of a neural network can be thought of as matrix calculations and activation functions. We will do some actual computations with matrices to see this in action.

Exercise

Provided below are the following:

- Three weight matrices `W_1`, `W_2` and `W_3` representing the weights in each layer. The convention for these matrices is that each $W_{i,j}$ gives the weight from neuron i in the previous (left) layer to neuron j in the next (right) layer.
- A vector `x_in` representing a single input and a matrix `x_mat_in` representing 7 different inputs.
- Two functions: `soft_max_vec` and `soft_max_mat` which apply the soft_max function to a single vector, and row-wise to a matrix.

The goals for this exercise are:

1. For input `x_in` calculate the inputs and outputs to each layer (assuming sigmoid activations for the middle two layers and soft_max output for the final layer.
2. Write a function that does the entire neural network calculation for a single input
3. Write a function that does the entire neural network calculation for a matrix of inputs, where each row is a single input.
4. Test your functions on `x_in` and `x_mat_in`.

This illustrates what happens in a NN during one single forward pass. Roughly speaking, after this forward pass, it remains to compare the output of the network to the known truth values, compute the gradient of the loss function and adjust the weight matrices `W_1`, `W_2` and `W_3` accordingly, and iterate. Hopefully this process will result in better weight matrices and our loss will be smaller afterwards

```
W_1 = np.array([[2,-1,1,4],[-1,2,-3,1],[3,-2,-1,5]])
W_2 = np.array([[3,1,-2,1],[-2,4,1,-4],[-1,-3,2,-5],[3,1,1,1]])
W_3 = np.array([[-1,3,-2],[1,-1,-3],[3,-2,2],[1,2,1]])
x_in = np.array([.5,.8,.2])
x_mat_in = np.array([[.5,.8,.2],[.1,.9,.6],[.2,.2,.3],[.6,.1,.9],[.5,.5,.4],[.9,.1,.9],[.1,.8,.7]])

def soft_max_vec(vec):
    return np.exp(vec)/(np.sum(np.exp(vec)))

def soft_max_mat(mat):
    return np.exp(mat)/(np.sum(np.exp(mat),axis=1).reshape(-1,1))

print('the matrix W_1\n')
print(W_1)
print('-'*30)
print('vector input x_in\n')
print(x_in)
print ('-'*30)
print('matrix input x_mat_in -- starts with the vector `x_in`\n')
print(x_mat_in)
```

```
the matrix W_1

[[ 2 -1  1  4]
 [-1  2 -3  1]
 [ 3 -2 -1  5]]
------------------------------
vector input x_in

[0.5 0.8 0.2]
------------------------------
matrix input x_mat_in -- starts with the vector `x_in`

[[0.5 0.8 0.2]
 [0.1 0.9 0.6]
 [0.2 0.2 0.3]
 [0.6 0.1 0.9]
 [0.5 0.5 0.4]
 [0.9 0.1 0.9]
 [0.1 0.8 0.7]]
```

Exercise

1. Get the product of array x_in and W_1 (z2)
2. Apply sigmoid function to z2 that results to a2
3. Get the product of a2 and z2 (z3)
4. Apply sigmoid function to z3 that results to a3
5. Get the product of a3 and z3 that results to z4

```
#W_1 = np.array([[2,-1,1,4],[-1,2,-3,1],[3,-2,-1,5]])
#W_2 = np.array([[3,1,-2,1],[-2,4,1,-4],[-1,-3,2,-5],[3,1,1,1]])
#W_3 = np.array([[-1,3,-2],[1,-1,-3],[3,-2,2],[1,2,1]])
#x_in = np.array([.5,.8,.2])
#x_mat_in = np.array([[.5,.8,.2],[.1,.9,.6],[.2,.2,.3],[.6,.1,.9],[.5,.5,.4],[.9,.1,.9],[.1,.8,.7]])


#1
z2 = np.multiply(W_1.T, x_in)
print(z2)
```

```
[[ 1.  -0.8  0.6]
 [-0.5  1.6 -0.4]
 [ 0.5 -2.4 -0.2]
 [ 2.   0.8  1. ]]
```

```
#2
a2 = sigmoid(z2)
print(a2)
```

```
[[0.73105858 0.31002552 0.64565631]
 [0.37754067 0.83201839 0.40131234]
 [0.62245933 0.0831727  0.450166  ]
 [0.88079708 0.68997448 0.73105858]]
```

```
#3
z3 = np.multiply(a2, z2)
print(z3)
```

```
[[ 0.73105858 -0.24802042  0.38739378]
 [-0.18877033  1.33122942 -0.16052494]
 [ 0.31122967 -0.19961447 -0.0900332 ]
 [ 1.76159416  0.55197958  0.73105858]]
```

```
#4
a3 = sigmoid(z3)
print(a3)
```

```
[[0.67503753 0.4383108  0.59565515]
 [0.45294706 0.79104392 0.45995472]
 [0.57718538 0.45026143 0.47750689]
 [0.8534092  0.63459475 0.67503753]]
```

```
#5
z4 = np.multiply(a3, z3)
print(z4)
```

```
[[ 0.49349198 -0.10871003  0.2307531 ]
 [-0.08550297  1.05306094 -0.0738342 ]
 [ 0.17963721 -0.0898787  -0.04299147]
 [ 1.50336067  0.35028335  0.49349198]]
```

```
def soft_max_vec(vec):
    return np.exp(vec)/(np.sum(np.exp(vec)))

def soft_max_mat(mat):
    return np.exp(mat)/(np.sum(np.exp(mat),axis=1).reshape(-1,1))
```

7. Apply soft_max_vec function to z4 that results to y_out

```
def softmax_vec(z):
    exp_z = np.exp(z - np.max(z))  # Subtracting the maximum value for numerical stability
    return exp_z / np.sum(exp_z)

# Assuming z4 is the input vector
y_out = softmax_vec(z4)

print(y_out)
```

```
[[0.08562215 0.04688707 0.06583853]
 [0.04798791 0.14983175 0.04855115]
 [0.0625577  0.04777839 0.05007193]
 [0.2350534  0.07419787 0.08562215]]
```

```
## A one-line function to do the entire neural net computation

def nn_comp_vec(x):
    return soft_max_vec(sigmoid(sigmoid(np.dot(x,W_1)).dot(W_2)).dot(W_3))

def nn_comp_mat(x):
    return soft_max_mat(sigmoid(sigmoid(np.dot(x,W_1)).dot(W_2)).dot(W_3))
```

```
nn_comp_vec(x_in)
```

```
array([0.72780576, 0.26927918, 0.00291506])
```

```
nn_comp_mat(x_mat_in)
```

```
array([[0.72780576, 0.26927918, 0.00291506],
       [0.62054212, 0.37682531, 0.00263257],
       [0.69267581, 0.30361576, 0.00370844],
       [0.36618794, 0.63016955, 0.00364252],
       [0.57199769, 0.4251982 , 0.00280411],
       [0.38373781, 0.61163804, 0.00462415],
       [0.52510443, 0.4725011 , 0.00239447]])
```

Backpropagation

The backpropagation in this part will be used to train a multi-layer perceptron (with a single hidden layer). Different patterns will be used and the demonstration on how the weights will converge. The different parameters such as learning rate, number of iterations, and number of data points will be demonstrated

```
#Preliminaries
from __future__ import division, print_function
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Fill out the code below so that it creates a multi-layer perceptron with a single hidden layer (with 4 nodes) and trains it via back-propagation. Specifically your code should:

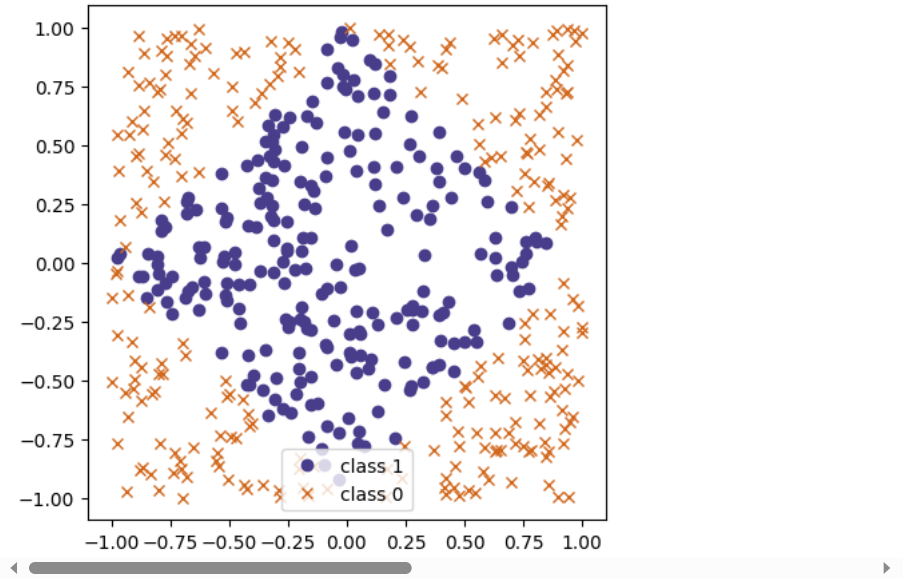
1. Initialize the weights to random values between -1 and 1
2. Perform the feed-forward computation
3. Compute the loss function
4. Calculate the gradients for all the weights via back-propagation
5. Update the weight matrices (using a learning_rate parameter)
6. Execute steps 2-5 for a fixed number of iterations
7. Plot the accuracies and log loss and observe how they change over time

Once your code is running, try it for the different patterns below.

- Which patterns was the neural network able to learn quickly and which took longer?
- What learning rates and numbers of iterations worked well?

```
## This code below generates two x values and a y value according to different patterns
## It also creates a "bias" term (a vector of 1s)
## The goal is then to learn the mapping from x to y using a neural network via back-propagation

num_obs = 500
x_mat_1 = np.random.uniform(-1,1,size = (num_obs,2))
x_mat_bias = np.ones((num_obs,1))
x_mat_full = np.concatenate( (x_mat_1,x_mat_bias), axis=1)

# PICK ONE PATTERN BELOW and comment out the rest.

# # Circle pattern
# y = (np.sqrt(x_mat_full[:,0]**2 + x_mat_full[:,1]**2)<.75).astype(int)

# # Diamond Pattern
y = ((np.abs(x_mat_full[:,0]) + np.abs(x_mat_full[:,1]))<1).astype(int)

# # Centered square
# y = ((np.maximum(np.abs(x_mat_full[:,0]), np.abs(x_mat_full[:,1])))<.5).astype(int)

# # Thick Right Angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>-.5)).astype(int)

# # Thin right angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>0)).astype(int)


print('shape of x_mat_full is {}'.format(x_mat_full.shape))
print('shape of y is {}'.format(y.shape))

fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
# ax.grid(True)
ax.legend(loc='best')
ax.axis('equal');
```

```
shape of x_mat_full is (500, 3)
shape of y is (500,)
<ipython-input-140-0f8bccfdade3>:32: UserWarning: color is redundantly defined by the 'c
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darksla
<ipython-input-140-0f8bccfdade3>:33: UserWarning: color is redundantly defined by the 'c
  ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocola
```



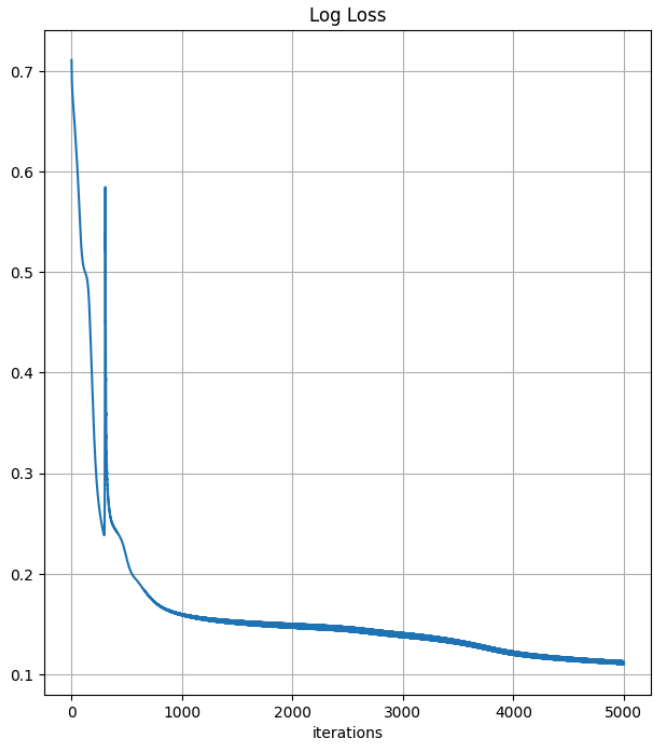
```
def sigmoid(x):
    """
    Sigmoid function
    """
    return 1.0 / (1.0 + np.exp(-x))


def loss_fn(y_true, y_pred, eps=1e-16):
    """
    Loss function we would like to optimize (minimize)
    We are using Logarithmic Loss
    http://scikit-learn.org/stable/modules/model_evaluation.html#log-loss
    """
    y_pred = np.maximum(y_pred,eps)
    y_pred = np.minimum(y_pred,(1-eps))
    return -(np.sum(y_true * np.log(y_pred)) + np.sum((1-y_true)*np.log(1-y_pred)))/len(y_true)


def forward_pass(W1, W2):
    """
    Does a forward computation of the neural network
    Takes the input `x_mat` (global variable) and produces the output `y_pred`
    Also produces the gradient of the log loss function
    """
    global x_mat
    global y
    global num_
    # First, compute the new predictions `y_pred`
    z_2 = np.dot(x_mat, W_1)
    a_2 = sigmoid(z_2)
    z_3 = np.dot(a_2, W_2)
    y_pred = sigmoid(z_3).reshape((len(x_mat),))
    # Now compute the gradient
    J_z_3_grad = -y + y_pred
    J_W_2_grad = np.dot(J_z_3_grad, a_2)
    a_2_z_2_grad = sigmoid(z_2)*(1-sigmoid(z_2))
    J_W_1_grad = (np.dot((J_z_3_grad).reshape(-1,1), W_2.reshape(-1,1).T)*a_2_z_2_grad).T.dot(x_mat).T
    gradient = (J_W_1_grad, J_W_2_grad)

    # return
    return y_pred, gradient


def plot_loss_accuracy(loss_vals, accuracies):
    fig = plt.figure(figsize=(16, 8))
    fig.suptitle('Log Loss and Accuracy over iterations')

    ax = fig.add_subplot(1, 2, 1)
    ax.plot(loss_vals)
    ax.grid(True)
    ax.set(xlabel='iterations', title='Log Loss')

    ax = fig.add_subplot(1, 2, 2)
    ax.plot(accuracies)
    ax.grid(True)
    ax.set(xlabel='iterations', title='Accuracy');
```

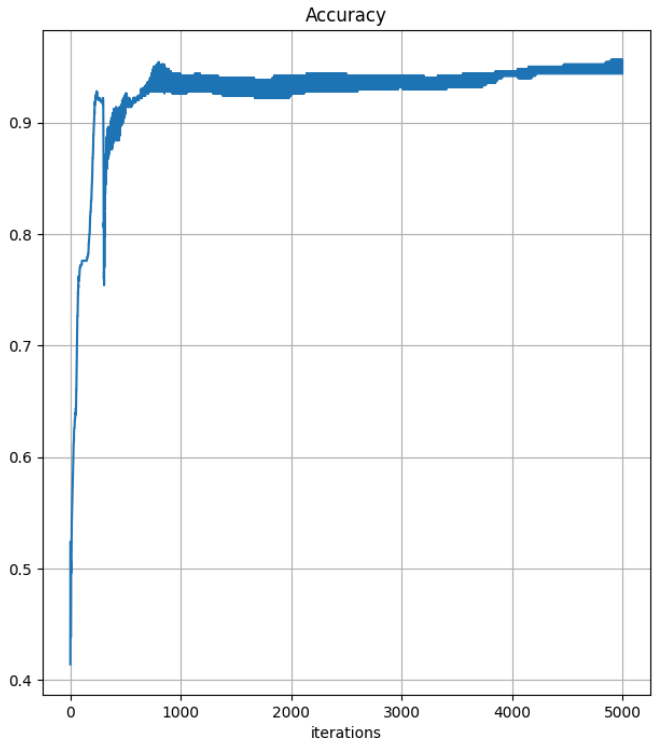
Complete the pseudocode below

```
# Initialize the network parameters
np.random.seed(1241)
W_1 = np.random.randn(3, 5)  # Assuming appropriate dimensions
W_2 = np.random.randn(5)      # Assuming appropriate dimensions
num_iter = 5000
learning_rate = 0.01
x_mat = x_mat_full
y = y.reshape((-1,))
loss_vals, accuracies = [], []

for i in range(num_iter):
    # Do a forward computation, and get the gradient
    y_pred, gradient = forward_pass(W_1, W_2)

    # Unpack gradients
    J_W_1_grad, J_W_2_grad = gradient

    # Update the weight matrices
    W_1 -= learning_rate * J_W_1_grad
    W_2 -= learning_rate * J_W_2_grad

    # Compute the loss and accuracy
    loss = loss_fn(y, y_pred)
    accuracy = np.mean(np.round(y_pred) == y)

    # Append loss and accuracy to lists
    loss_vals.append(loss)
    accuracies.append(accuracy)

    # Print the loss and accuracy for every 200th iteration
    if i % 200 == 0:
        print(f"Iteration {i}: Loss - {loss}, Accuracy: {accuracy}")

plot_loss_accuracy(loss_vals, accuracies)
```

```
Iteration 0: Loss - 0.7107837201802948, Accuracy: 0.414
Iteration 200: Loss - 0.3482639492467614, Accuracy: 0.858
Iteration 400: Loss - 0.24335562970341865, Accuracy: 0.886
Iteration 600: Loss - 0.19236532584975702, Accuracy: 0.922
Iteration 800: Loss - 0.16835325804126208, Accuracy: 0.928
Iteration 1000: Loss - 0.16015641190704405, Accuracy: 0.926
Iteration 1200: Loss - 0.15645287121196733, Accuracy: 0.928
Iteration 1400: Loss - 0.15421370067257664, Accuracy: 0.924
Iteration 1600: Loss - 0.15265458348655245, Accuracy: 0.924
Iteration 1800: Loss - 0.1514280126009721, Accuracy: 0.922
Iteration 2000: Loss - 0.15034799456808934, Accuracy: 0.926
Iteration 2200: Loss - 0.1492673487993543, Accuracy: 0.928
Iteration 2400: Loss - 0.1479818730388399, Accuracy: 0.928
Iteration 2600: Loss - 0.1460806779413562, Accuracy: 0.928
Iteration 2800: Loss - 0.14355288813508046, Accuracy: 0.93
Iteration 3000: Loss - 0.14128510010975853, Accuracy: 0.932
Iteration 3200: Loss - 0.13886064441640444, Accuracy: 0.93
Iteration 3400: Loss - 0.13575516293484788, Accuracy: 0.93
Iteration 3600: Loss - 0.13178383768759955, Accuracy: 0.932
Iteration 3800: Loss - 0.12700148040635215, Accuracy: 0.936
Iteration 4000: Loss - 0.12282290540138283, Accuracy: 0.942
Iteration 4200: Loss - 0.1199038524227915, Accuracy: 0.942
Iteration 4400: Loss - 0.11772777686919442, Accuracy: 0.944
Iteration 4600: Loss - 0.11596266636950071, Accuracy: 0.944
Iteration 4800: Loss - 0.11443511080301384, Accuracy: 0.944
```

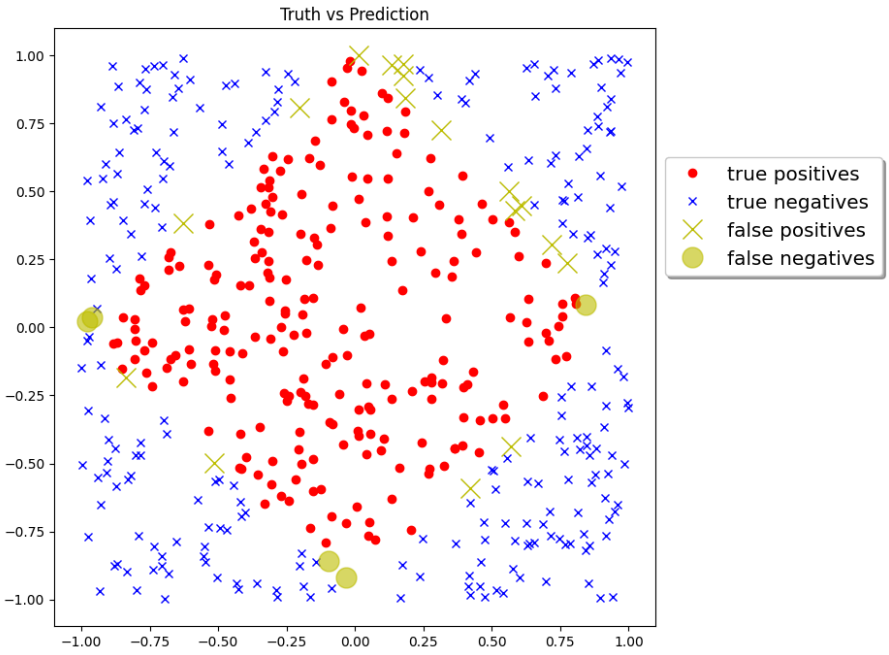
Log Loss and Accuracy over iterations





Plot the predicted answers, with mistakes in yellow

```
pred1 = (y_pred>=.5)
pred0 = (y_pred<.5)

fig, ax = plt.subplots(figsize=(8, 8))
# true predictions
ax.plot(x_mat[pred1 & (y==1),0],x_mat[pred1 & (y==1),1], 'ro', label='true positives')
ax.plot(x_mat[pred0 & (y==0),0],x_mat[pred0 & (y==0),1], 'bx', label='true negatives')
# false predictions
ax.plot(x_mat[pred1 & (y==0),0],x_mat[pred1 & (y==0),1], 'yx', label='false positives', markersize=15)
ax.plot(x_mat[pred0 & (y==1),0],x_mat[pred0 & (y==1),1], 'yo', label='false negatives', markersize=15, alpha=.6)
ax.set(title='Truth vs Prediction')
ax.legend(bbox_to_anchor=(1, 0.8), fancybox=True, shadow=True, fontsize='x-large');
```



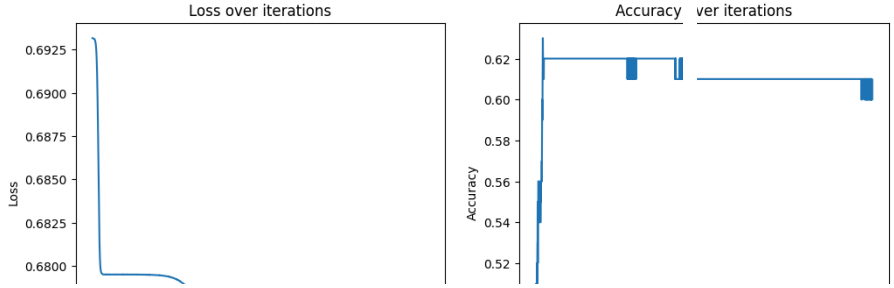
Supplementary Activity

1. Use a different weights , input and activation function
2. Apply feedforward and backpropagation
3. Plot the loss and accuracy for every 300th iteration

Double-click (or enter) to edit

```
# ReLU activation function
def relu(x):
    return np.maximum(0, x)

# Derivative of ReLU
def relu_derivative(x):
    return np.where(x > 0, 1, 0)

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

# Generate random dataset
np.random.seed(0)
X = np.random.rand(100, 2)  # 100 samples, 2 features
y = np.random.randint(0, 2, (100, 1))  # Binary classification problem

# Initialize weights with smaller random values
W1 = np.random.randn(2, 10) * 0.01  # Increase neurons in the first laye
W2 = np.random.randn(10, 1) * 0.01  # Weights for the second layer

# Learning rate
learning_rate = 0.1  # Increase learning rate

# Number of iterations
num_iter = 5000  # Increase number of iterations

# Loss and accuracy tracking
loss_vals, accuracies = [], []

# Training loop
for i in range(num_iter):
    # Feedforward
    z1 = np.dot(X, W1)
    a1 = relu(z1)
    z2 = np.dot(a1, W2)
    y_pred = sigmoid(z2)

    # Compute loss
    loss = -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
    loss_vals.append(loss)

    # Compute accuracy
    accuracy = np.mean((y_pred > 0.5) == y)
    accuracies.append(accuracy)

    # Backpropagation
    delta2 = y_pred - y
    dW2 = np.dot(a1.T, delta2 * sigmoid_derivative(z2))
    delta1 = np.dot(delta2 * sigmoid_derivative(z2), W2.T)
    dW1 = np.dot(X.T, delta1 * relu_derivative(z1))

    # Update weights
    W1 -= learning_rate * dW1
    W2 -= learning_rate * dW2

    # Print loss and accuracy every 300th iteration
    if i % 300 == 0:
        print(f"Iteration {i}: Loss - {loss}, Accuracy - {accuracy}")

# Plot loss and accuracy
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(loss_vals)
plt.title('Loss over iterations')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(1, 2, 2)
plt.plot(accuracies)
plt.title('Accuracy over iterations')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
```

```
Iteration 0: Loss - 0.6931555346580736, Accuracy - 0.48
Iteration 300: Loss - 0.6795124397658224, Accuracy - 0.62
Iteration 600: Loss - 0.6795055467301004, Accuracy - 0.62
Iteration 900: Loss - 0.6794866397492649, Accuracy - 0.62
Iteration 1200: Loss - 0.6793137771487783, Accuracy - 0.62
Iteration 1500: Loss - 0.6783467748680012, Accuracy - 0.62
Iteration 1800: Loss - 0.6769313114756578, Accuracy - 0.62
Iteration 2100: Loss - 0.6765349752328281, Accuracy - 0.61
Iteration 2400: Loss - 0.6765050469642546, Accuracy - 0.61
Iteration 2700: Loss - 0.6764895032313063, Accuracy - 0.61
Iteration 3000: Loss - 0.6764915057083059, Accuracy - 0.61
Iteration 3300: Loss - 0.6764897797574878, Accuracy - 0.61
Iteration 3600: Loss - 0.6764911712922275, Accuracy - 0.61
Iteration 3900: Loss - 0.6764800127414484, Accuracy - 0.61
Iteration 4200: Loss - 0.6764503161370011, Accuracy - 0.61
Iteration 4500: Loss - 0.6762715230230022, Accuracy - 0.61
Iteration 4800: Loss - 0.6758107675999055, Accuracy - 0.61
```



ReLU activation function

def relu(x): return np.maximum(0, x)